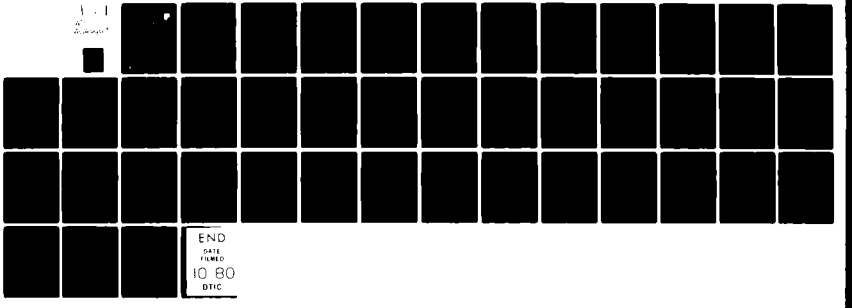
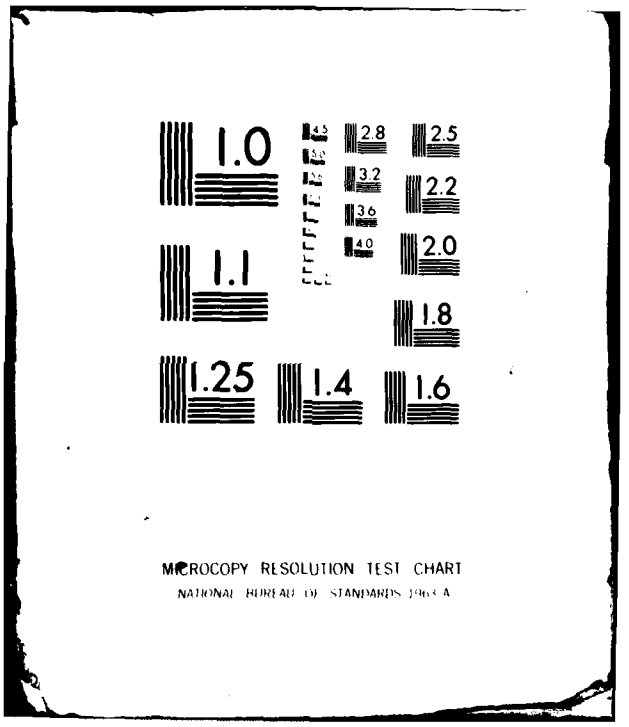
AD-A089 997 POLYTECHNIC INST OF NEW YORK BROOKLYN F/G 9/2
SOFTWARE TEST MODELS AND IMPLEMENTATION OF ASSOCIATED TEST DRIV--ETC(U)
MAR 80 D L BAGGI, M L SHOOMAN F30602-78-C-0057
UNCLASSIFIED POLY-EE-79-0057 RADC-TR-80-45 NL



END
DATE FILMED
10 80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

LEVEL II (12)

RADC-TR-80-45
Interim Report
March 1980

SOFTWARE TEST MODELS AND IMPLEMENTATION OF ASSOCIATED TEST DRIVERS

Polytechnic Institute of New York

Dennis L. Baggi
Martin L. Shooman

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

APPROVED: Alan N. Sukert

ALAN N. SUKERT
Project Engineer

APPROVED: Wendall C. Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-80-45	2. GOVT ACCESSION NO.: AD-A089997	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): SOFTWARE TEST MODELS AND IMPLEMENTATION OF ASSOCIATED TEST DRIVERS.	5. TYPE OF REPORT & PERIOD COVERED: Interim Report. Jan 78 - Jun 79
	6. PERFORMING ORG. REPORT NUMBER: Poly-EE-79-0057
7. AUTHOR(s): Dennis L. Baggi, Martin L. Shooman	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0057
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Polytechnic Institute of New York, 333 Jay Street, Brooklyn NY 11201	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2304J401, J4
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: Mar 1980
	13. NUMBER OF PAGES: 43
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Alan Sukert (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Test Models
Software Test Drivers
Automatic Software Testing
Forced Execution

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In the past, most software tests were constructed by heuristics and by drawing upon experience with similar software. Recently, enough preliminary work has been done to propose an analytical construction of test cases.

This report begins by defining five broad classes of software tests: Type 0, Type 1, Type 2, Type 3 and Type 4. In a Type 0 test, all instructions are exercised at least once. In a type 1 and 2 test, all

(Cont'd) next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

43 204

Item 20 (Cont'd)

cont.

flowchart paths are exercised at least once. Type 1 is performed by forced traversal and Type 2 by natural execution. Types 3 and 4 correspond to an exhaustive interaction of all INPUT AND STORED DATA. Clearly, Types 3 and 4 are unfeasible and only a strategy lying between Type 1 and 2 can effectively be implemented.

Since enumeration of all the paths in a given program is required for Type 1 and 2 tests, this report establishes the lower and upper bounds on the number of paths as a function of the number of deciders, describes a manual decomposition procedure to cut a graph into smaller subgraphs, and proposes an algorithm to machine-identify all paths. A complete Type 1.5 driver system for forced path traversal, implemented in PL/1, is then thoroughly described, together with suggestions on how to extend these techniques to other languages.

A typical program is analyzed manually, tested with data and run through the system. Some evaluation of the usefulness of the system is eventually given in the light of the accumulated experience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAM LISTINGS

1.0 Introduction

At the present state of the programming art, there exist two techniques for removing errors from a program during the various stages of development, program proofs and program testing. Although much effort has been expended on program proofs, it is not clear whether this method will become a practical and widely used technique. The present universally used technique is to test to remove bugs, either by code reading, by walkthroughs, or by machine testing.

To investigate a strategy for testing -- be it manual, semi-automatic or automatic --it is necessary to provide some theoretical background, such as formal definitions and analytic models, to fully define the range and scope of the test project. In general, it is indeed unclear what really is meant by error models, debugging procedures, and other such terms. We describe here a hierarchy of testing models. The importance of testing cannot be exaggerated, because only a well-tested program can be assumed to be reasonably error-free, in the prevailing lack of general techniques to prove the correctness of procedures.

Much of the testing presently done is ad hoc and heuristic rather than having any theoretical background. The purpose of this report is to present some models and analytical techniques which can be used in developing software test systems. It will be shown that practical driver systems for automatic testing can be implemented from formal definitions of testing types.

The test type to be discussed in detail is a Type 1 test, which is defined as a test model in which each program path is force-traversed once. The definition involves a discussion on how program branching points and loops affect the number of paths. The process culminates in an algorithm for identifying all program paths.

The possibility of implementing and automating such a testing model is then investigated. It is shown that the technique is feasible; a system of programs has been implemented to force execution through all possible paths of a given program under test. This requires that the system analytically determine all program paths from the code, modify the input code and drive several runs of the program. Study of these forced runs will result in many program errors being caught without having to calculate and insert particular testing data, a major effort if done by hand for a complex program. The computer output for each run contains a unique labeling of the path traversed, related error messages and normal output[1], if any, and the amount of time elapsed during that run. The system has already proved itself very valuable in program debugging on a few problems, with its fully automatic mode of operation being the significant asset.

Section 2 is a short survey of similar efforts for automated testing systems. Section 3 defines in detail a hierarchy of test types; Section 4

[1]Note that forced testing may result in program outputs which differ from those produced by natural testing; however, these can be readily identified by the tester.

deals with the analytical determination of paths in program flowcharts; Section 5 describes in detail driver systems and associated algorithms, and Section 6 considers the results and limitations of the system. Finally, Section 7 concludes by considering the advantages and disadvantages of the models, and a proposal for future research efforts.

2.0 Drivers for Testing - A Brief Survey

The idea of automatic drivers for software testing is certainly as old as the discipline of software engineering. Quite a few models for testing have been proposed in the past, based on techniques ranging from some form of dynamic program analysis to automatic data generation for traversal of program paths (4,5,6,7,8,10,12,16,17). We shall briefly discuss some of these methods and some of the advantages and disadvantages inherent in these techniques.

The execution of a program may, in general, be described by a process of mapping a set of input data values into some output data by the use of some intermediate, internal data. Testing may therefore be accomplished by assigning some critically chosen values to the input data, for which the output values may be known in advance, and by running the program to check for consistency. It subsequently becomes clear that the technique could be extended to what is generally referred to as Symbolic Execution, a form of generalized testing (4,7). In such a case, a program is said to be executed "symbolically" if symbols are introduced as input values replacing real data objects (such as integers and reals). In trivial cases involving no symbols, the process would be identical to normal execution. The extension of normal execution to symbolic is analogous to the extension of numeric arithmetic to symbolic algebraic operations. Hence, during symbolic execution of the program, a variable has a fixed but unknown value, and therefore one single run is equivalent to a large class of manual test runs. Assignment of values to all symbols would correspond to normal execution, and furthermore, between these two extremes, a tester may choose an intermediate strategy of assigning a value only to some variable symbols.

During a symbolic run, computation of symbolic expressions is generally delayed, or generalized. Conditional expressions are handled by exploring both the "true" and the "false" branch (conflicts may be resolved at some latter point). One symbolic execution run may be characterized by an "execution tree" and further applied to testing. Based upon this strategy, a system called EFFIGY has been developed, which algebraically represents a program path's computation by symbolically executing it (6,7).

SELECT (4) is another driver system based on the technique of symbolic execution. It handles all paths of a given program by symbolically traversing all of them and by constructing all input and intermediate data necessary to exercise that path and produced in it. It operates on a LISP-like language, i.e., a subset of LISP to which constructs such as FOR, WHILE and UNTIL have been added. A path may be exercised by "forward substitution" of values within the deciders, values which are stored in a LISP-list (this list contains, at the end of the run, all the values corres-

ponding to a given path); and by "backward substitution," which consists of resubstituting the values causing a run through a desired path. Conditions causing branching in a program are inequalities and equalities. A subprogram attempts to solve the system of inequalities with a conjugate gradient ("hill-climbing") algorithm; obviously a certain solution corresponds to a certan path. The system further allows the user to include assertions about the program as an adjunct to the program code, whose consistency with the program can be proved or disproved by the mechanism. One clear advantage of this strategy is that it finds and excludes from execution all unreachable paths. A minor criticism can be stated by remarking that the language handled by the system is a subset of LISP with some ALGOL constructs, which probably has the disadvantages of both languages without the advantages of neither and which, in any case, only moderately approximates the coding techniques used in the real world of programming. Furthermore, the inequality solver's hill-climbing algorithm is not guaranteed to work in general and requires human interaction, thus preventing the system from being fully automated.

A similar methodology can be used to automatically generate data for path testing, as described in (5). From symbolic executon one can derive a set of constraints on the values of the input data set. The method described in (5) consists, among others, of some preprocessing of the subject program to be tested, of the generation of some data base from it, and of translation of the subject program into some intermediate code for symbolic execution. Path selection can be static (meaning automatic generation of paths) or interactive (under user's control). Data determination is achieved by first simplifying the resulting inequalities and then solving them. The inequality solver attempts a solution and adds the constraints one by one, checking for consistency. If the previous solution still holds, it is retained, otherwise a new one is generated. The main disadvantage of this ingenious strategy is that it is impossible to solve a general system of inequalities. A serious restriction is that path analysis must operate with linear inequalities. The system handles FORTRAN programs, but with a few restrictions (e.g., array references dependent on input values are not allowed). In spite of the restrictions, however, the application of path analysis techniques to a real language has considerable merit.

Some other strategies for automatic testing have been proposed, among which we shall consider the type of driver system described in (8). The system works for FORTRAN programs and operates upon code segments, defined as a set of consecutive statements to which control may be transferred (presumably a construct corresponding to a compound statement in programming languages with structured features). Segment relationship depends upon how the flow of control transfers from one segment to another (probably a definition rendered necessary by the wide use of GO TO's in FORTRAN). The driver then attempts to traverse an optimal path for testing. Identification of segments, relationships, type of branching, etc. is achieved through syntactic analysis. A base path is then generated as a concatenation of segments, and finally a path optimizer selects paths and the order of execution of the code. Thus the system can automatically supply the tester with an analysis chart of a given program, and the tester

is responsible for the execution of the path. The system has been implemented and completed with practical applications in mind; hence its use is geared toward FORTRAN and some language-dependent constructs. As a result, one may voice the well known reservations deriving from the FORTRAN-versus-structured languages controversy.

Another set of automatic software test drivers is described in (12). Their main objective is the construction of test cases for the proper execution and linkage of calling and called routines in terms of external interface of target-program modules. IBM's Automated Unit (16) works with a low-level assembler-like language, Module Interface Language - Specific (MIL-S), and creates a test procedure in MIL-S from a FORTRAN segment. Other systems operate directly at source level, performing operation of test procedures with the goal of assisting the tester by managing test data and automatically running simple tests.

We will show below that our system employs a totally new approach. Although it is implemented in PL/1 and handles PL/1 programs, the techniques can be extended with almost no effort to any other structured language, such as PASCAL, C, (C is the Bell Labs language in which UNIX is written), ADA, etc., and with some further research even to FORTRAN and assemblers. Its main features consist of a static analysis of the source code determining the program structure, and of a dynamic part in which the program is force-executed through all its paths. The main disadvantage of the strategy is that some normally unreachable paths may be reached by the system. However, the approach allows the implementation of a fully automatic system requiring no human interaction and guaranteed to explore all paths of a program for any case.

3.0 Types of Tests

3.1 Introduction

We shall begin with a formal definition of various types of testing strategies. We shall note that, in devising a classification scheme for testing models, it is natural to desire that it correspond to an increasing (or decreasing) hierarchy of thoroughness and difficulty. Clearly, the upper range of our numerical scheme should correspond to an exhaustive test. At the lower end of the range we will require only that each instruction be executed at least once.

We might liken the types of tests to the test procedures which an owner might apply to check a new car he has just purchased from a dealer. The first, and most rudimentary, check would be to compare the list of accessories he ordered wth the delivered list on the car window, and see if these are present and work. For example, the owner might check to see that he got an AM/FM radio, and that it works on both AM and FM; that he received a V-6 engine and not a straight six or a V-8; that the engine starts; the hood lamp, glove box lamp, and trunk lamp were installed and work; etc. This check list type test would be the lowest level. At the other extreme would be functional testing, i.e., use of the auto for three

months. However, in between, he would try many things during his first week of driving: drive the car up a hill with and without the air conditioner on, try the heater on a cool day and the air conditioner on a hot one (or alternate the two functions), accelerate from rest to 60 mph and try a panic stop, etc.

Thus, the philosophy for test classification which we will use applies to product testing in general. However, the specific details will apply to software in particular.

3.2 Completeness and Continuity Checking - Type 0

This type of testing requires that each instruction be exercised at least once.

Intuition tells us that in testing a mechanism one basic principle is to try and exercise the parts. In the case of a program, such a test is very much expedited by a modern assembler or compiler whose location counter assigns a number to each instruction or statement. A Type 0 test is a necessary but not sufficient condition for thorough testing of the program. In fact, when such a test is employed, one often finds design flaws. For example, it is sometimes impossible to reach a section of code, and upon detailed investigation, one finds that an error was corrected by inserting a patch to bypass a block of code. However, the block was never removed and just remains inert.

Obviously, a Type 0 test can be performed at the module level as well as at the system integration level. It is more common to allow the individual coder (or tester) freedom at the module stage to proceed as he wishes. Thus, much of our definition of test types is more applicable to integration testing.

A common way to implement a Type 0 test is to exercise each function at least once, and check the code, pseudocode, or flow chart to see which code is checked out. For example, in a word processing system we might check to see that the editor can be reached, that each editor function works, etc. At this stage of testing, it is unnecessary to check interactions of features; i.e., we don't have to enter the editor system to change a word, store it on disk, and then recall the new version. However, even such a low level test requires a great deal of effort and bookkeeping in a large system, unless a computerized tool is developed as an aid. Some practitioners have even suggested that a machine architectural feature be added which reserves a machine word bit for such checking. A special instruction would be added to zero all these bits initially, and whenever a machine instruction is executed, the respective check bit could be set to one. Thus, a memory dump (or a search for nonset bits) could be used to reveal which sections had not been tested.

3.3 All Paths Force-Executed - Type 1

One of the problems in testing a program at a level higher than zero is the dependence between the data and the decider predicates (expressions which control the branching of an IF-THEN-ELSE or DO WHILE instruction) in the program. Intuition tells us that once we have completed a check list for a Type 0 test, we should next test all paths in the program. If we use a flowchart as our program abstraction, we can define all paths of the chart. However, it is unfeasible to determine by manual analysis all possible executable paths in most programs. Thus, an automated tool is highly desirable.

In the solution to the problem of constructing a program testing tool, it is convenient to define two classes of path tests: Force-Execution, Type 1, and Natural-Execution, Type 2. By natural execution we mean that the tester (human or machine) reads the decider predicates, computes whether they are true or false based on the current values of the program variables, and branches left or right accordingly. This concept applies also to 3-way or multi-way branching, because such constructs can always be expressed by 2-way branches. Modern IF-THEN-ELSE constructs express indeed this fact that a condition is either true or false. To simplify the problem, we have defined the artificial concept of forced-execution (1,2). In forced-execution the tester only recognizes the fact that it has reached a decider as it progresses through the program. Once it discovers a decider it forces further execution of the program for two cases, one where the decider is true and one where the decider is false.

We should however mention that such a model has a flaw. It is obvious that forced execution will traverse some unfeasible paths, while real data would prevent natural execution from reaching particular sections, as in the example of Fig. 3.1. Forced execution would traverse path A-B, thus causing overflow in branch B. However, natural execution of branch A prevents subsequent execution of branch B. On the other hand, it is equally true that some real errors, appearing only with natural-execution, will never be detected by forced-execution. This is true for any driver model, regardless of its level of sophistication. Even with these limitations, Type 1 testing provides a quick and inexpensive technique for detecting many program errors, because the benefits of automatic analysis outweigh the disadvantages of some unnatural test cases.

The execution time of a program is often largely devoted to the repetitive execution of DO loops within the program. However, the philosophy of a forced test is to execute all paths which only include, at most, two executions of a program loop. Thus, we must invent a technique to ensure that each DO loop is traversed no more than twice. We also know from experience that many errors are committed when we exit from a loop. Thus, we define forced execution of a DO loop as testing the loop twice, one for the first value of the index and again for the last value. Methods of forced execution of paths and of DO loops are discussed in Section 4.

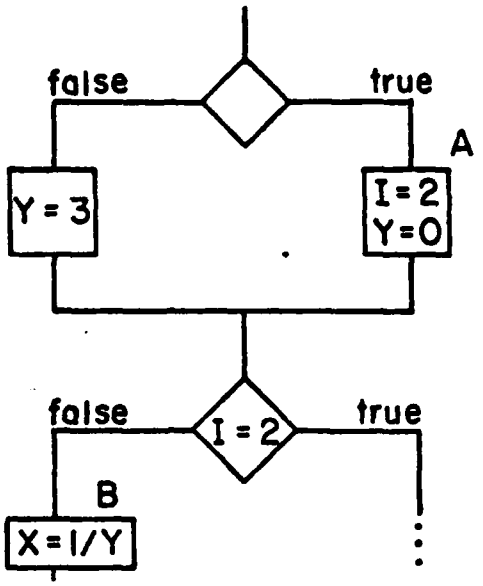


Figure 3.1. Unreachable Path. Path A-B is never reached in natural-execution mode.

Another question relevant to force-traversal methodology is whether or not input data have to be supplied at execution. Obviously, data whch affect the flow of control are not needed and can be omitted. However, other data types, such as operands in arithmetic expressions, may profitably be submitted if the user is interested in such testing. Therefore, it essentially depends upon whether or not the user of force-traversal testing merely wishes to check the control flow and path structure for which no input data are necessary. If he'd like to test for consistency of results, he must supply input values. At any rate, a Type 1 driver can run without any input data whatsoever, and it is this fully automatic mode of operation that renders the model so attractive. The fact that some program variables may contain meaningless quantities is but a natural limitation of this type of model, and can be solved only by an escalation to a higher testing model. Nevertheless, the present strategy greatly enhances the user's access to thorough testing of programs.

3.4 All Paths Naturally Executed - Type 2

The next highest level of program testing is to let a program naturally-traverse all its paths, defined as a Type 2 test. The practical implementation of such a strategy requires access to a set of test data where each member of the set causes one run of the program through a particular path. This data set could be designed by the user, clearly a major effort with unreliable results, or by the use of some computer assisted tools, as described in Section 2. However, there is no known fully automatic pro-

cedure for the generation of such data, and thus an automatic Type 2 driver is not feasible. Furthermore, for each member of the data set, one can find, in general, an infinity of data exercising the same path. Thus a Type 2 test is not unique, but rather represents a class of tests.

We suggest that a realistic strategy is made possible by a test model of Type 1.5, i.e., between Types 1 and 2. According to such a model, some selected paths would be naturally executed with their associated data, and an exhaustive test would be completed by force-traversal of all other paths to ensure complete coverage.

3.5 Exhaustive Testing - Types 3 and 4

Similarly, we define here two types of exhaustive tests. If we assume that neither the input nor stored data are inherently probabilistic, and we wish to construct an exhaustive test, we must now test each path not only for one value of input data per path (as for Type 2), but for the entire range of combnations allowable. The problem of calculating the number of needed combinations reduces to a combinatorial problem yielding a huge number of cases. In all practical cases, it is unfeasible to perform an exhaustive test, and since a Type 2 may not be thorough enough, a realistic implementation of a Type 3 test is possible only by resorting to some heuristically constructed test which is somewhere between Type 2 and 3. If we use heuristics to choose the number of test cases, we should make sure that the variable ranges include cases of positive, negative and zero values, as well as other values which traditionally cause trouble.

If either the stored system status data or input data is probabilistic in nature, the sequence variables must also be included in computing an exhaustive list of combinations. This is a Type 4 test and it differs from Type 3 in that additional sequence variables are needed to define an exhaustive test. Again, the number of combinations in an exhaustive test renders the model unfeasible. It is clear that a Type 4 test is the upper limit for testing, which includes all other types as special cases.

Table 3.1 summarizes the class definitions which we have evolved, and discusses one typical "in between" classification, Type 1.5.

4.0 Analytical Determination of Program Paths

4.1 Introduction

In this section we analyze the relationship between the number of decider predicates in a loopless program and the number of program paths.

First, an upper and lower bound are determined in Section 4.2. Then a decomposition procedure is explained in Section 4.3, and an example is given which shows how all possible paths in a program flowchart can be identified from its structure.

TABLE 3.1 -- Classification of Tests	
Types	Discussion
0	All instructions in code executed at least once (check list).
1	All paths force-executed at least once (simulated 100% coverage).
1.5	All paths force-executed, some naturally executed.
2	All paths naturally-executed at least once (path coverage 100%). This test is not unique.
3	All paths naturally-executed for all values of input parameters (exhaustive test).
4	All paths naturally-executed for all values of input parameters, all sequences of inputs, and all combinations of initial conditions (exhaustive test for multiprocessing, multiprogramming, and real time systems with non-fixed input sequence).

4.2 Bounds on the Number of Paths in a Loopless Program

The important properties of flowcharts are:

(1) the number of decision elements (deciders);

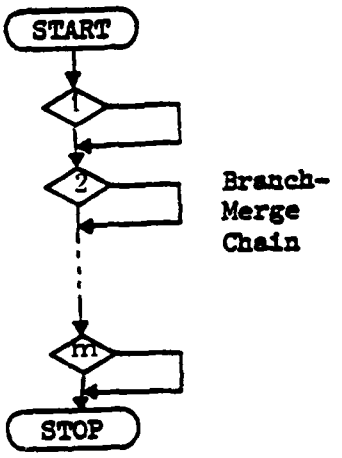
(2) the number of points where two or more feed forward branches meet (merges);

(3) the number of points where a feed forward path meets a feedback path and creates a loop.

At each of these points one can write a simple equation relating each path. Repeated use of these relations leads to the analytic determination of the number of paths in a flowchart.

For simplicity we assume that the flowgraph has no loops. We attempt to bind the number of paths to the number of deciders and merges. In Figure 4.1(a) we show a graph with m deciders and m merges. Each decider-merge pair furnishes two paths. By virtue of the chain structure, we see that the number of paths for the total graph is simply the product of each subgraph path, i.e., 2^m paths. In Figure 4.1(b) we portray a structure with n deciders and one merge. The first decider creates two paths.

The next decider takes up one of the paths as its input and creates two new paths. Thus, there are n+1 paths in this graph. As an example of the application of these bounds, consider a graph with 13 deciders. The number of paths in such a graph is between 14 and 8192. Our intuition and experience with some examples seems to point out that the number of paths in a program is usually closer to the lower bound.



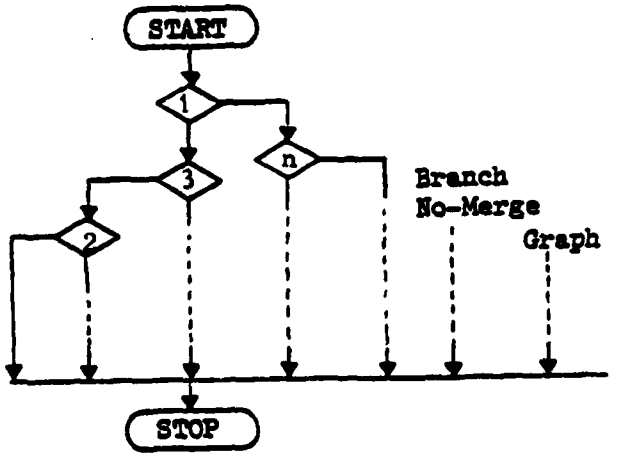


Fig. 4.1(a) An Upper Bound. Fig. 4.1(b) A lower bound.

FIGURE 4.1. Flowchart Bounds on the Number of Paths.

4.3 Procedure for Manual Determination of the Number of Paths

If a program is written in structured top-down form or any other modular form, the program can easily be divided into independent sub-graphs. In the case of a nonstructured design, subdivision can still be performed with analogous techniques.

In performing subdivisions, the elementary sub-structures given in Figure 4.2 are encountered. In Figure 4.2(a) the number of paths in the program between point A and stop or stops is denoted by N_A. Clearly this number is the sum of the number of paths attached to the left hand branch N_B and those attached to the right hand branch N_C. In Figure 4.2(b) the branch-merge-structure multiplies the number of paths seen at point B by 2 whereas in the case of Figure 4.2(c) we end up with two equalities at the merge, as shown.

We will illustrate the calculation of the number of paths in a program with n conditional jumps. From the previous discussion, we know this number to be in the range $(n+1, 2^n)$. Let us now consider the following example:

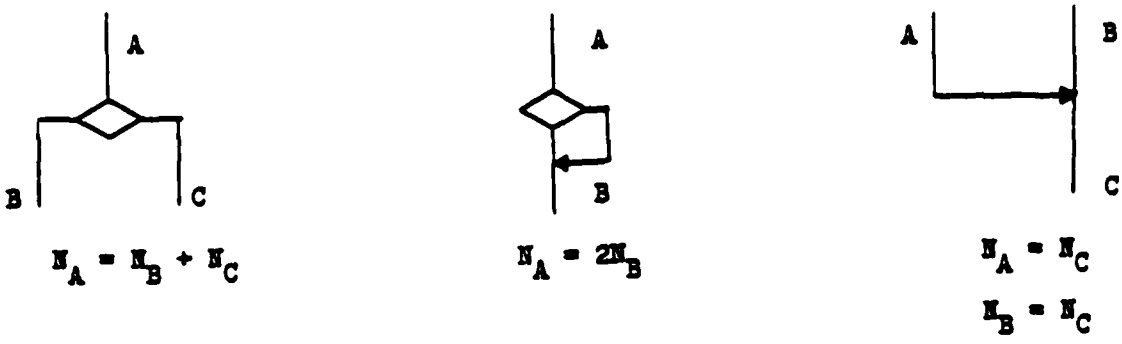


Fig. 4.2(a) Branch. Fig. 4.2(b). Branch-Merge Fig. 4.2(c). Merge

FIGURE 4.2. Elementary Graph Sub-Structures

Assume that the computer is to determine the winner of a card game in which player A is dealt two cards: A1, A2, and similarly player B is dealt two cards, i.e., B1, B2. If the players have any pairs, the highest pair wins; otherwise, the player with the highest card wins. If both players have the same high card, then the winner is the player with the highest second card. Identical hands with or without pairs are ties. A flowchart for this program is given in Figure 4.3. There are 13 deciders, and each branch is identified with letters A, A', B, etc. The flowchart is decomposed in sub-modules labeled A, A', B, B', etc., as shown in Figure 4.3 and the simple algebraic relationships which can be derived are listed in Table 4.1. All paths are identified and taken into account one by one; the final computation for this structure with 13 deciders yields 100 paths.

We now have an analytic technique for the manual determination of the number of flowchart paths. The procedure is, however, time consuming and error prone even for very simple cases. It is desirable to realize a fully automatic algorithm to machine-identify all paths (not simply count them); such a programmable algorithm is described in the next section.

5.0 Driver Systems

5.1 Introduction

We will now introduce the practical implementation of Type 1.5 driver systems. Such drivers force the traversal of a given subject program through all its paths.

Recall that if we naturally execute a subset of all program paths, then we refer to such a test as being between Type 1 and Type 2. Similarly, in most cases, forced-execution will coincide or can be made to coincide with natural-traversal of some paths and forced-traversal of the remainder or can be made as such. Consequently, we describe the drivers discussed here as Type 1.5 tests.

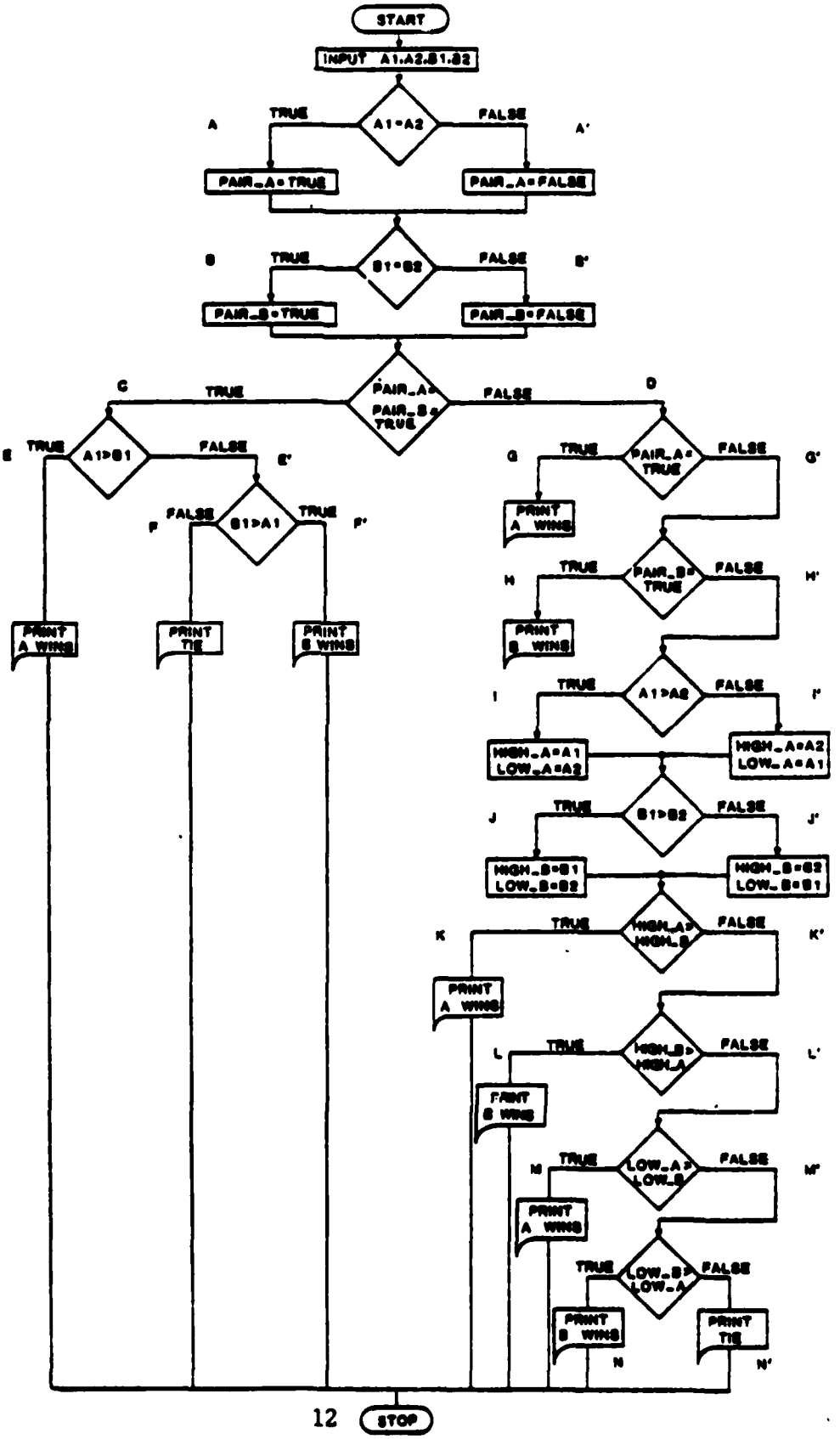


Figure 4.3

Flowchart for Computer Solution of a Card Game

TABLE 4.1 Number of Paths Calculated in the Flowchart of Figure 4.3	
Algebraic Relationship	Number of Paths
$N = N_A + N_{A'} = 2 \times N_{A'}$	$2 \times N_{A'}$
$N_{A'} = N_B + N_{B'} = 2 \times N_{B'}$	$4 \times NB'$
$N_{B'} = N_C + N_D$	$4 \times (N_C + N_D)$
$N_C = N_E + N_{E'} = 1 + N_{E'}$	$4 \times ((1 + N_{E'}) + N_D)$
$N_{E'} = N_F + N_{F'} = 1 + 1 = 2$	$4 \times (3 + N_D)$
$N_D = N_G + N_{G'} = 1 + N_{G'}$	$4 \times (3 + 1 + N_{G'})$
$N_{G'} = N_H + N_{H'} = 1 + N_{H'}$	$4 \times (4 + 1 + N_{H'})$
$N_{H'} = N_I + N_{I'} = 2 \times N_{I'}$	$4 \times (5 + 2 \times N_{I'})$
$N_{I'} = N_J + N_{J'} = 2 \times N_{J'}$	$4 \times (5 + 4 \times N_{J'})$
$N_{J'} - N_K + N_{K'} = 1 + N_{K'}$	$4 \times (5 + 4 \times (1 + N_{K'}))$
$N_{K'} = N_L + N_{L'} = 1 + N_{L'}$	$4 \times (5 + 4 \times (1 + 1 + N_{L'}))$
$N_{L'} = N_M + N_{M'} = 1 + N_{M'}$	$4 \times (5 + 4 \times (2 + 1 + N_{M'}))$
$N_{M'} = N_N + N_{N'} = 1 + 1 = 2$	$4 \times (5 + 4 \times (3 + 2))) = 100$

The design of drivers has evolved through several phases during the present research on testing models. The most obvious technique for complete path traversal is referred to as an "upper bound" driver and is described in Section 5.2. Such a design, as it will be shown, achieves the goal of automated path testing at a high penalty. Further considerations and refinements of the problem, namely, the realization of an algorithm for path analysis, have led to the implementation of a system of programs which constitute the whole driver system. These will be described in Sections 5.3 to 5.6.

5.2 An "Upper Bound" Driver

The system described here was a first attempt to implement a driver to force the execution of a PL/1 program under test, from now on referred to as the subject program.

The subject program is written in standard PL/1 with no restrictions. There are only a few precautions the programmer must take in designing his code:

- The total number of IF-statements and repetitive DO-groups, herein called NTESTS, must be supplied on a data card;
- Each statement of the form: IF cond ... must be written as
  IF F(cond) ...
- Each statement of the form: DO I=limit1 TO limit2 BY increment
  must be written as: DO I=GL(limit1, limit2) to GH BY increment
- Each statement of the form: DO WHILE(cond)
  must be written as: DO WHILE(H(cond))
- Functions and subroutines must be internal.

The deck of the subject program is then simply inserted within the deck of the driver program at an appropriate location. The driver exercises all paths through several runs.

The driver's mode of operation is simply based on the fact that the upper bound on the number of possible paths is 2^{NTESTS} (see Section 4.2). The driver program will internally construct a binary number, called control word, with NTESTS bits, whose initial value has all bits set to 0. This number is increased by 1 at each run during execution, till the control word has all bits set to 1.

At each run, function F (as well as GL, GH and H) replaces the value of the condition with the corresponding bit from the control word. Functions GL and GH cause a DO-group with an index variable to be executed once with the initial value of the index (bit=0), and once with the final value (bit=1). Function H causes execution of a DO WHILE group exactly once in any case.

Since there are 2^{NTESTS} possible distinct values of the control word, there will be exactly 2^{NTESTS} runs of the subject program. Therefore, the coverage of all possible paths is mathematically guaranteed. Hence, the goal of automated force-traversal is fully achieved with this simple strategy.

Because the number of paths in a program may be closer to the lower bound NTESTS+1 than to the upper bound 2^{NTESTS}, there will often be a large number of runs which do not represent any existing paths. For instance, the flowchart of Figure 4.3 has 13 deciders but only 100 paths; hence 8092 runs are wasted with this strategy. Furthermore, since the number of runs increases exponentially with the number of deciders, the running cost of such a driver becomes very prohibitive, even for medium size programs.

This problem can be overcome by the derivation of the path structure of a program from its code using static analysis. This strategy will be described in the following sections.

5.3 A Type 1.5 Driver

The complete driver system is shown in Figure 5.1. It has a section for static path analysis, one for code translation and one for dynamic testing. At the left hand side in the picture, one recognizes the execution of the driver programs from files located in the middle of the picture. An input program to be tested, referred from now on to as the subject program, enters the path analyzer, which determines the program paths and saves their representation as binary path descriptors, along with a copy of the subject program. This is described in Section 5.4.

The copy of the subject program undergoes some modifications performed by the translator program. This translator modifies conditional branches, loop constructs and includes the program in a large loop. This inclusion allows repeated execution. This is described in Section 5.5.

Eventually, the modified subject program reaches the execution stage through all its paths, as determined by the binary path descriptors, and the output of the driver is produced, as described in Section 5.6.

Although we have chosen to implement a PL/1 driver, it can be shown that these techniques are applicable to almost any language. For this particular implementation, we assume that the program is structured, contains no GO TO's and has been compiled successfully. The language PL/1 has been chosen because it is widely available and allows the design of well structured programs, since it possesses constructs such as IF-THEN-ELSE, DO WHILE, compound statements and blocks.

In spite of these restrictions, we will show that similar techniques for the construction of the driver can be applied to almost any language, structured or not. Any language possessing blocks, "if-then-else", and "while" constructs can be handled exactly as is PL/1, with the proper (isomorphic) change of syntax. This is the case for ALGOL, PASCAL, and C (a language developed at Bell Labs running on PDP-11 machines) and ADA. Furthermore, languages like C, LISP, etc., where an assignment statement can be embedded within a conditional expression (conveniently for our purposes, this is not the case for PL/1), may be adapted to our technique simply by isolating that statement from those affecting the control flow.

Languages which do not have structured programming constructs can be handled by the algorithm in another way. This is the case for FORTRAN and assembly languages. Note that considerable effort is spent today in writing structured FORTRAN and assemblers. Furthermore, there exist some structured versions of FORTRAN, such as ratfor (developed at Bell Labs), involving a preprocessor which converts if-then-else, while, etc.,

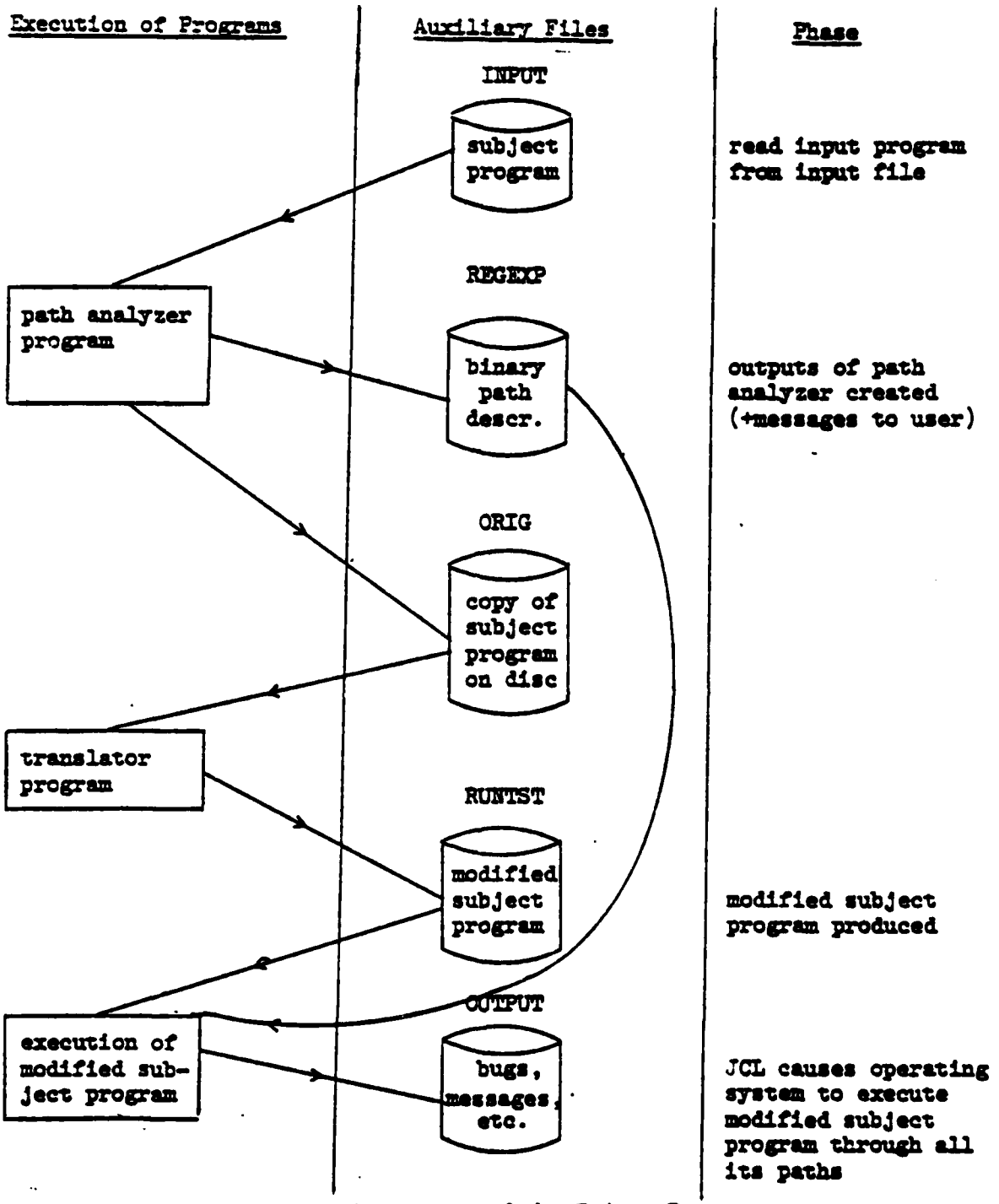


FIGURE 5.1 The System of the Driver Programs.

This shows the relationship between (a) execution of programs, (b) the auxiliary files, and (c) their phase of operations.

constructs into standard FORTRAN. A preprocessor which converts unstructured into structured FORTRAN has also been developed (Bell Labs). It is conceivable that similar techniques can be applied to programs written in assembly language. Therefore the approach could be used for any language (with the possible exception of LISP and SNOBOL). We are currently examining extensions of the technique to assembly and machine languages. Hence, the vast majority of computer users could benefit from these techniques.

5.4 The Algorithm for Path Analysis

5.4.1 Labeling of Paths: We shall use the convention to label the "true" branch of a conditional statement with a "1", and the "false" branch with a "0", as seen in Figure 5.2. In this way, it is possible to uniquely label a path in a given program with a binary path description, as shown in Figure 5.3.

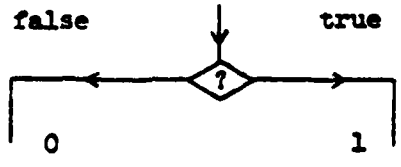


FIGURE 5.2 Labeling of branches.

5.4.2 Algorithm for Finding All Possible Paths: We will first show that it is possible to determine all possible paths. We will start by considering path analysis for a program without any repetitive DO constructs. Since each path is uniquely defined by a binary integer, referred to here as path descriptor, the problem of finding each path in a given program is analogous to the problem of finding the set of binary integers associated with the path structure of that program. Because sets of binary quantities can be expressed by regular expressions, we propose an algorithm which constructs a regular expression whose associated set contains the values of binary control words corresponding to the paths. Only the operations "+" (expressing union in the associated set) and concatenation will be needed. The expression is recursively defined as being always binary, i.e., it contains two terms separated by "+". A term is the symbol 1, or 0, or an expression; concatenation of expressions forms an expression.

The algorithm scans a PL/1 program in search of IF-THEN-ELSE constructs, and operates according to the following rules:

1. Each IF opens a left parenthesis, and initiates an expression.

2. Each THEN corresponds to a "1".

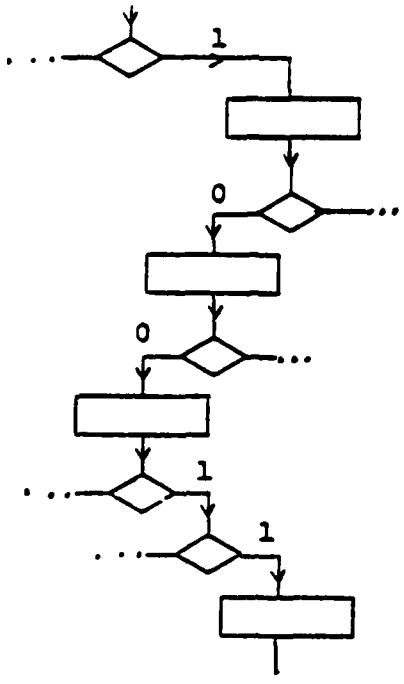


FIGURE 5.3. Labeling of Path 10011.

3. Each ELSE corresponds to a "0", and since it matches a previous THEN, a "+" is inserted at that level.

4. If no matching ELSE is present, it is assumed to be there and "+0" is added.

5. Each balanced expression, consisting of "1", "+", "0", closed at its level, causes closure with a right parenthesis at that level.

The following examples 1 to 4 contain a variety of flowchart constructs designed to illustrate how the algorithm works for most common programming segments. Each example consists of a few lines of PL/1 pseudo-code (i.e., in which only keywords such as IF, THEN, ELSE, DO, END are important) which are represented in the flowgraph of the associated figure. The reader may attempt to directly apply to the code the appropriate rule from the set of the above five rules. This will yield the regular expression listed below the code. Each element in the resulting expression is labeled underneath with the particular algorithm rule producing that element (examples of elements: "(", "1", "+0"). The regular expression could be solved by hand to determine its associated set of binary numbers, shown at the end of each example. These are the binary path descriptors. The reader can verify with the flowgraph that they indeed correspond to the paths in the program segment.

EXAMPLE 1.

Consider the flowchart of Figure 5.4. This chart is implemented by the program segment

```
IF cond THEN s
        ELSE s
IF cond THEN s
IF cond THEN s
        ELSE s
```

The algorithm constructs:

(1+0)(1+0)(1+0), applying algorithm rules
12 3512 4512 35

Computation of the regular expressions yields

111,011,101,001,110,010,100,000

i.e., the eight possible paths.

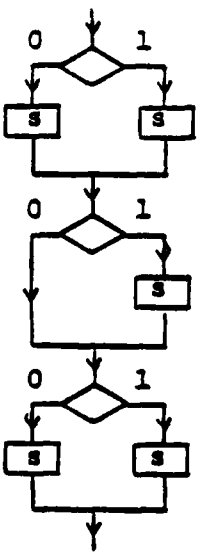


FIGURE 5.4. Flowchart for Example 1.

EXAMPLE 2.

The flowchart of Figure 5.5 translates into:

```
IF cond THEN IF cond THEN IF cond THEN s
                                      ELSE s
                          ELSE IF cond THEN s
                                      ELSE s
               ELSE IF cond THEN IF cond THEN s
                                      ELSE s
                          ELSE IF cond THEN s
                                      ELSE s
```

Regular expression:

(1(1(1+0)+0(1+0))+0(1(1+0)+0(1+0))), rules:
121212 35 312 355 31212 35 312 3555

representing the eight paths

111,110,101,100,011,010,001,000

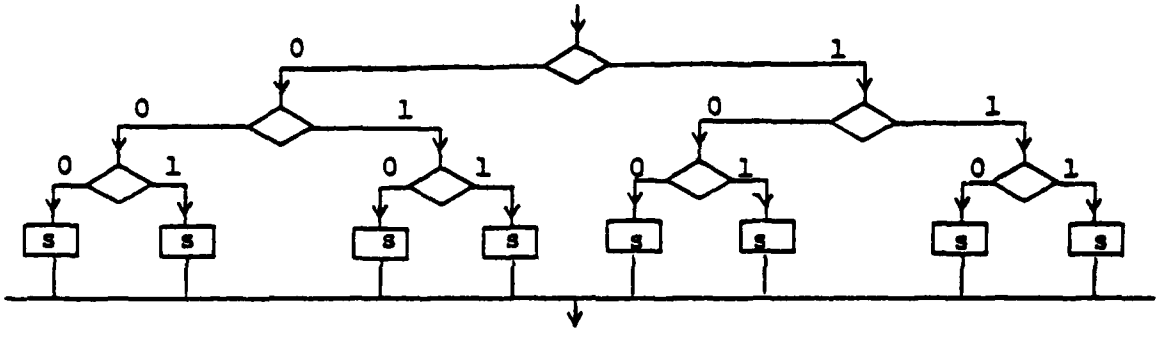


FIGURE 5.5. Flowchart for Example 2.

EXAMPLE 3.

The flowchart of Figure 5.6 is programmed by

```
IF cond THEN s
       ELSE IF cond THEN s
                  ELSE IF cond THEN s
                             ELSE IF cond THEN s
                                        ELSE s
```

Regular expression:

(1+0(1+0(1+0(1+0)))), rules:
12 312 312 312 35555

which gives

1,01,001,0001,0000

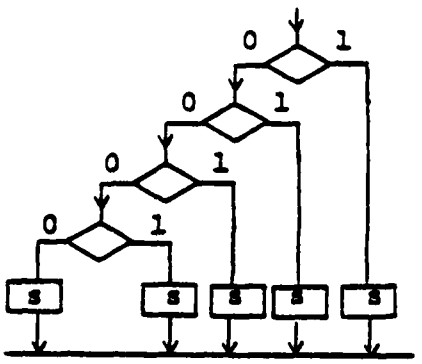


FIGURE 5.6. Flowchart for Example 3.

EXAMPLE 4.

The flowchart of Figure 5.7 is realized by

```
IF cond THEN DO;
     IF cond THEN IF cond THEN s
                          ELSE s
             ELSE IF cond THEN s
                          ELSE s
     IF cond THEN s
     END;
        ELSE DO;
     IF cond THEN;
             ELSE s
     END;
```

Regular expression:

(1(1(1+0)+0(1+0))(1+0))+0(1+0)), rules:
121212 35 312 355 12 45 312 355

yielding

1111,1110,1101,1100,1011,1010,1001,1000,01,00

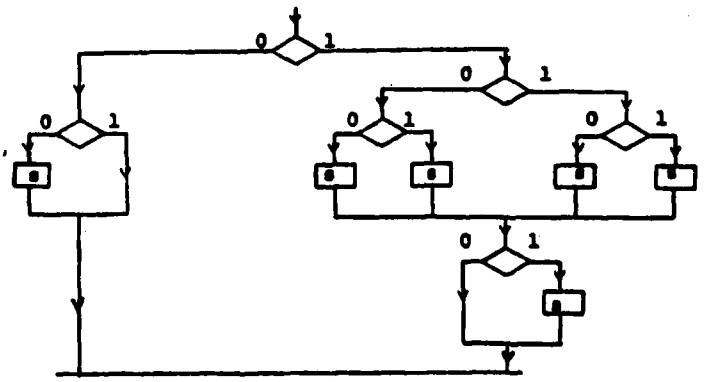


FIGURE 5.7. Flowchart for Example 4.

In example 4, the path structure is complicated by the presence of DO groups within the THEN and ELSE clauses. When this occurs, construction of the regular expression temporarily halts, and the algorithm calls itself recursively for that DO module. This algorithm has been tested with a program written in LISP (it's easier to design a recursive algorithm in that language) and then translated into PL/1.

A second algorithm solves the regular expression and finds all the elements of its associated set. The number of elements is the number of all possible paths. Hence the algorithm, as an extra bonus, enumerates all paths of a program.

The complete analyzing program embodies these algorithms. It starts by reading in the subject program, card by card. This is stored as a character string in the main memory, and is saved on an external file, called ORIG (see Figure 5.1). Control is then passed to the section which performs the scanning and computes the set of binary control words.

The scanning mechanism is built around two PL/1 procedures, CAR and CDR, which respectively return the head and the tail of a given string. By head we mean: Any PL/1 operator such as *, **, ¬, >, ||), any PL/1 separating character (such as ;, :, (,)), any string separated by an operator, a separating character or a blank, or any quoted string or comment; by tail we mean the string without its head. Hence, CAR is capable of correctly selecting keywords in portions of statements, such as ;IF(,*/THEN/*, L2:DO;, but will not return those in THEN1=5, 'A STRING IF NEEDED', /*THEN A COMMENT*/.

The analyzing program produces the set of binary path descriptors (stored on external file REGEXP, see Figure 5.1), which will control the successive phases of the subject program execution performed by the driver system.

We are ready to consider the example of Figure 4.3, described in Section 4.3. The path analyzer program of the driver system would produce the output shown in Listing 1. The reader can convince himself that each of the 100 binary path descriptors corresponds to an existing path in Figure 4.3. The regular expression corresponds to the structure of the flowchart, and its associated set yields all paths.

5.5 Program Translator for Forced Driving

This second program reads the subject program from file ORIG and performs a few PL/1 string manipulations, to transform it into a program able to run through all its paths.

It inserts at the top of the program some declarations of auxiliary variables and procedures which were not in the original program; these entities will control the test runs. A simple statement is inserted in order to allow continued execution of the new object program even if it runs out of data, which are not used anyhow in most runs.

HERE IS THE PROGRAM TO ANALYZE:

```
 PROGRAM:PROC OPTIONS(MAIN);
         DCL(A1,A2,B1,B2,HIGH_A,LOW_A,HIGH_B,LOW_B)FIXED BINARY INIT(0),
             (PAIR_A,PAIR_B)BIT(1);
              GET LIST(A1,A2,B1,B2)COPY;
         IF     A1=A2      THEN PAIR_A='1'B;ELSE PAIR_A='0'B;
         IF     B1=B2      THEN PAIR_B='1'B;ELSE PAIR_B='0'B;
         IF     PAIR_A & PAIR_B      THEN
                IF     A1>B1  THEN PUT SKIP LIST('A WINS');
                           ELSE IF A1<B1 THEN PUT SKIP LIST('B WINS');
                                         ELSE PUT SKIP LIST('TIE');
                           ELSE S1:DO;
         IF     PAIR_A     THEN PUT SKIP LIST('A WINS');
    ELSE IF     PAIR_B     THEN PUT SKIP LIST('B WINS');
                       ELSE S2:DO;
         IF     A1>A2     THEN  DO;HIGH_A=A1;LOW_A=A2;END;
                          ELSE  DO;HIGH_A=A2;LOW_A=A1;END;
         IF     B1>B2     THEN  DO;HIGH_B=B1;LOW_B=B2;END;
                          ELSE  DO;HIGH_B=B2;LOW_B=B1;END;
         IF     HIGH_A>HIGH_B     THEN PUT SKIP LIST('A WINS');
    ELSE IF     HIGH_B>HIGH_A     THEN PUT SKIP LIST('B WINS');
    ELSE IF     LOW_A>LOW_B       THEN PUT SKIP LIST('A WINS');
    ELSE IF     LOW_B>LOW_A       THEN PUT SKIP LIST('B WINS');
                                  ELSE PUT SKIP LIST('TIE');
                   END S2;  END S1;
 END PROGRAM;
```

THE CORRESPONDING REGULAR EXPRESSION IS:

```
((1)+(0)((1)+(0)((1((1)+(0((1)+(0)))))+(0((1)+(0((1)+(0((1)+(0)((1)+(0)((1)+(0((1)+(0((1)+(0((1)+(0)))))))))))))))
```

GENERATING THE BINARY WORDS:

```
1111,11101,11100,1101,11001,11000111,11CC0101,1100C1001,11000110001,1100011000C,11000101,110001001,1100010001,11000100
001,110001000C,11000011,110000101,1100001001,11000C10001,11000010000,11000001,110000001,1100000001,11000000001,11000000
000,1011,10101,101C0,1001,10001,10000111,100001101,10J0011001,100CC110001,1000C110000,10000101,100001001,1000010001,1000
0100001,100001C0000,10000011,100000101,1000001001,10000010C01,10000010000,1000000l,100000001,1000000001,10000000001,1000
0000000,0111,01101,C1100,0101,01001,010C0111,010001101,010CC110C1,01000110001,01000110000,01000101,010001001,0100010001,
01000100001,010001000000,C1000011,010000101,0100001001,01000010001,01000010000,01000001,010000001,0100000001,01000000001,
01000000000,0011,001C1,001C0,0C01,C0001,000C0111,00000111C1,000CC110C1,00000110001,00000110000,00000101,000001001,0000010
001,00000100001,0J000100000,00000011,000000101,0000001001,00000010001,00000010000,00000001,000000001,0000000001,00000000
001,0000000000C
```

Listing 1. Program listing for flowchart of Figure 4.3. The corresponding regular expression and binary path descriptors.

The original program is embedded into an infinite loop. Its execution and exit will be controlled by the system-created function @WORD, which reads the binary path descriptor to select the path and stops the program when the test is completed.

The object program is scanned in search of conditional statements and repetitive DO groups.

Any construction of the form

```
IF cond THEN ...
```

is replaced by

```
IF @F (cond) THEN ...
```

where @F is a procedure function replacing, at running time, the original value of the condition with the corresponding value within the binary path descriptor representing the path being examined.

It is the responsibility of the translator to modify the subject program to force traversal of each repetitive DO-group at least once, and not more than twice. The following cases can occur at execution time:

- The values of initial value, final value and increment of the control variable are such that the loop would be skipped altogether. Consistent with the strategy of traversing all possible paths, such a loop will be executed once, regardless.
- The initial value equals the final value; the loop is executed once.
- Otherwise, the loop is executed twice; once with the index, or control variable, set to its initial value, and once with the highest (or smallest) value of the index variable for which the loop is still executable. *Example: in the loop DO I=0 TO 9 BY 2, the final value of the index* variable is 8 (not 9).

This is accomplished by transforming any construct of the form

```
DO CV=IV TO FV BY INC WHILE(cond);
```

or

```
DO CV=IV BY INC TO FV WHILE(cond);
```

(where IV, FV, INC are, in general, expressions representing the initial value, final value and increment, and CV is the control variable) into:

```
DO CV=IV TO FV BY @G(CV,IV,FV,INC) /*WHILE(cond)*/;
```

If the BY clause is missing in the original, INC is set to 1 in the argument list of @G, and if the TO clause is missing FV is set to IV. @G is a function which returns a value for the increment. This value will control execution of the loop exactly in the manner described above, forcing execution once or twice, as the case dictates.

Recall that there is always a binary path descriptor associated uniquely with each path and that each bit of the binary word is the value of the expression in the corresponding IF-statement in that path. However, since it is possible that there are conditional statements within a repetitive DO-group, it becomes necessary to store the current scanning position within the binary path descriptor when a loop is entered the first time and to resume scanning from that same position when the second execution of the loop begins. Otherwise the system would, so to speak, fall out of synchronism, picking up a bit value which would not anymore correspond to the conditions it should be assigned to. This is accomplished by two functions: @H1 saves the current scanning position on a stack, whenever a loop executable twice is entered. @H2 pops that value from the stack upon re-entry of the same loop. Obviously the stack level corresponds to the level of loop nesting. In this way, care is taken that the proper order of scanning a path descriptor is maintained.

Lastly, notice that since the construct

```
DO WHILE(cond);
```

becomes

```
DO /*WHILE(cond)*/;
```

this iterative DO group is executed just once, regardless of the value of the condition.

Hence, the object program has been transformed to let it run as many time as there are paths, with proper action enforced at the loop. And finally, this new program is sent to the external file RUNTST, which is then called into PL/1 execution by the operating system (see Figure 5.1). As an example we will consider the following simple program segment, consisting of a repetitive DO-group:

```
DO I=3 TO BY 34-J*2 TO L1 WHILE(X**2¬=64);
     X=X+1;
     IF PATHBRANCH=1 THEN PUT LIST('HERE');
                     ELSE PUT LIST('THERE');
END;
```

The translator program would write on file RUNTST the following lines of code:

```
DO I=3 TO L1 BY @G(I,3,L1,34-J*2) /* WHILE(X**2¬= 64)*/;
      IF I=3 & @G(I,3,L1,34-J*2) <=L1-3 THEN CALL @H1;
      IF I¬=3 THEN CALL @H2;
      X=X+1;
      IF @F(PATHBRANCH=1) THEN PUT LIST('HERE');
                          ELSE PUT LIST('THERE');
END;
```

The segment has been modified. @G will return a new increment value to force the loop to execute at most twice. Then @H1 is called only the first time the loop is entered (I=3), and only if the loop executes twice (new increment = final value - initial value) and pushes the path descriptor scanning position on the stack. @H2 is called only when the loop is executed the second time around (I¬=3). Finally, @F will force traversal of the path described by the corresponding bit in the path descriptor.

As a second and final example, we shall show the translated code (see Listing 2) for the program of Listing 1 whose flowchart is shown in Figure 4.3. As of the present, it is not necessary for the reader to follow in detail the operation of the modified subject program. He can, however, readily recognize that the original code is embedded in the new program (with some automatically added indentation to make DO-groups more readable), and that any identifier whose first character is # (for variables) and @ (for procedures) has been produced by the translator. The TIMER functions are external assembly language subroutines connected to the inner clock; all other procedures are internal.

5.6 Execution of the Translated Subject Program

The original object program is inserted into an infinite loop. Control of its execution is assumed by a function procedure @WORD (see Listing 2). This routine reads in the binary path descriptor belonging to that path from file REGEXP (see Figure 5.1).

Function @F is invoked each time a conditional statement is met at execution. It fetches the next bit from the control binary word and assigns that value to the condition, hence forcing the flow of control through a branch, regardless of the original value of the condition.

Function @G returns the increment value of a repetitive DO-group forcing execution at most twice, but at least once. The present scanning position within the controlling path descriptor is saved in a stack. An external procedure written in assembly language is used to compute the execution time in a path.

The output produced by each run contains information relevant to that path. The number of the run, or of the path, is printed together with its associated path descriptor. Hence, the reader can see by inspection which program path caused trouble, if any. The run catches and prints errors caused by interrupts, such as end-of-file, overflow and underflow, division by 0, etc. It then prints the time elapsed in each path. However, provi-

```
STMT LEVEL NEST

  1                 @P:PROC OPTIONS(MAIN);
  2     1           DCL #T BIT(1000)VAR,                                          (#PAT
                    HS,#BITPOS)FIXED BIN INIT(1),                          @F ENTRY(BIT(1))
                    RETURNS(BIT(1));
  3     1           DCL #BPD(256)FIXED,#I FIXED INIT(1);
  4     1           DCL (#TLIMIT,#FL,#TLEFT)FIXED BIN (31);
  5     1           DCL #TUSED FLOAT;
  6     1           DCL @G ENTRY(FIXED,FIXED,FIXED,FIXED)RETURNS(FIXED);
  7     1           DCL REGEXP FILE INPUT STREAM;
  8     1           OPEN FILE(REGEXP);
  9     1           ON ENDFILE(SYSIN);
 11     1           ON ENDFILE(REGEXP)STOP;
 13     1           CALL @WORD;
 14     1           #TLIMIT=5000;
 15     1           @LOOP:DO WHILE(1);
 16     1     1     CALL @TIMER(#TLIMIT,#FL);

 17     1     1           PROGRAM:BEGIN;
 18     2     1           DCL ( A1 , A2 , B1 , B2 , HIGH_A , LOW_A , HIGH_B , LOW_B ) FIXED BINARY IN
                    IT ( 0 ) , ( PAIR_A , PAIR_B ) BIT ( 1 ) ;
 19     2     1           GET LIST ( A1 , A2 , B1 , B2 ) COPY ;
 20     2     1           IF @F( A1 = A2) THEN PAIR_A = '1'B ;
 22     2     1           ELSE PAIR_A = '0'B ;
 23     2     1           IF @F( B1 = B2) THEN PAIR_B = '1'B ;
 25     2     1           ELSE PAIR_B = '0'B ;
 26     2     1           IF @F( PAIR_A & PAIR_B) THEN IF @F( A1 > B1) THEN PUT SKIP LIST ( 'A WINS'
                    ) ;
 29     2     1           ELSE IF @F( A1 < B1) THEN PUT SKIP LIST ( 'B WINS' ) ;
 31     2     1           ELSE PUT SKIP LIST ( 'TIE' ) ;
 32     2     1           ELSE S1 : DO;
 33     2     2                IF @F( PAIR_A) THEN PUT SKIP LIST ( 'A WINS' ) ;
 35     2     2                ELSE IF @F( PAIR_B) THEN PUT SKIP LIST ( 'B WINS' ) ;
 37     2     2                ELSE S2 : DO;
 38     2     3                     IF @F( A1 > A2) THEN DO;
 40     2     4                          HIGH_A = A1 ;
 41     2     4                          LOW_A = A2 ;
 42     2     4                          END;
 43     2     3                     ELSE DO;
 44     2     4                          HIGH_A = A2 ;
 45     2     4                          LOW_A = A1 ;
 46     2     4                          END;
 47     2     3                     IF @F( B1 > B2) THEN DO;
 49     2     4                          HIGH_B = B1 ;
 50     2     4                          LOW_B = B2 ;
 51     2     4                          END;
 52     2     3                     ELSE DO;
 53     2     4                          HIGH_B = B2 ;
 54     2     4                          LOW_B = B1 ;
 55     2     4                          END;
 56     2     3                     IF @F( HIGH_A > HIGH_B) THEN PUT SKIP LIST ( 'A WINS' ) ;
 58     2     3                     ELSE IF @F( HIGH_B > HIGH_A) THEN PUT SKIP LIST ( 'B WINS' ) ;
 60     2     3                     ELSE IF @F( LOW_A > LOW_B) THEN PUT SKIP LIST ( 'A WINS' ) ;
 62     2     3                     ELSE IF @F( LOW_B > LOW_A) THEN PUT SKIP LIST ( 'B WINS' ) ;
```

Listing 2. Translated code for the program of Listing 1. (Part 1)

```
STMT LEVEL NEST

 64     2    3                  ELSE PUT SKIP LIST ( 'TIE' ) ;
 65     2    3                  END S2;
 66     2    2              END S1;
 67     2    1          END PROGRAM;


 68     1    1    CALL STIMER(#TLEFT);
 69     1    1    #TUSED=(#TLIMIT-#TLEFT)/100E0;
 70     1    1    PUT SKIP EDIT('TIME ELAPSED ',#TUSED)(A,F(10,2));
 71     1    1    CALL &WORD;
 72     1    1    END &LOOP;
 73     1         CLOSE FILE(REGEXP);
 74     1         &F:PROC(S)   RETURNS(BIT(1));
 75     2         DCL 'S,S1) BIT(1);
 76     2         S1=SUBSTR(#T,#BITPOS,1);
 77     2         #BITPOS=#BITPOS+1;
 78     2         RETURN(S1);
 79     2         END &F;
 80     1         &WORD:PROC;
 81     2         GET FILE(REGEXP) LIST(#T);
 82     2         PUT SKIP(2) EDIT('PATH NO. ',#PATHS,'   PATH DESCRIPTOR:',
                   #T)(A,F(10),A,B);
 83     2         #PATHS=#PATHS+1;
 84     2         #BITPOS=1;
 85     2         END &WORD;
 86     1         &G :PROC(CV,IV,FV,INC)RETURNS(FIXED);
 87     2         DCL(CV,IV,FV,INC)FIXED;
 88     2         IF IV=FV THEN RETURN(1);
 90     2         IF INC=0 THEN RETURN(FV-IV);
 92     2         IF IV>FV & INC>0 THEN RETURN(FV-IV-1);
 94     2         IF IV<FV & INC<0 THEN RETURN(FV-IV+1);
 96     2         IF INC>0 THEN RETURN(FV-IV-MOD(FV-IV,INC));
 98     2         ELSE RETURN(FV-IV+MOD(IV-FV,-INC));
 99     2         END &G;
100     1         &H1:PROC;
101     2         #BPD(#I)=#BITPOS;
102     2         #I=#I+1;
103     2         END &H1;
104     1         &H2:PROC;
105     2         #I=#I-1;
106     2         #BITPOS=#BPD(#I);
107     2         END &H2;
108     1         END &P;
```

Listing 2. Translated code for the Program of Listing 1. (Part II)

sion is made, such that if a path's running time exceeds a certain threshold (as defined by the programmer), infinite loops will not run indefinitely and will be stopped. Finally, the output expected in executing that path appears on the printout.

Since a listing of the modified subject program is available to allow interpretation of the results, the user can make best use of the output by skipping those errors associated with unreachable paths or those derived from unspecified data, considering those errors which point to a structural error in the algorithm, identifying the corresponding path and fixing the code.

Note that the running mode does not require any explicit care to be taken by the programmer on the tested subject (except for a few restrictions noted in the next section). The insertion of the proper variable and procedures allowing exhaustive testing is automatically done by the system, first by the analyzer which constructs the path descriptors, and later by the translator which modifies the code. Hence, the programmer need not be aware of the transformations his code went through. To allow easy tracing, however, the original conditions have not been deleted from the code, and WHILE clauses are enclosed in comments. Thus, the original program could be executed normally and is still readable even in its new form on the listing.

We will now show, as an example of the result of running a translated subject program, the output for the program shown in Listings 1 and 2 and Figure 4.3. Recall that, because of the construction of the path analyzing algorithm (Section 5.4.1), the first paths identified and traversed will be those labeled ABCE, ABCE'F, ABCE'F' and ABDG in Figure 4.3. Hence the output will be as shown in Listing 3:

```
PATH NO.          1   PATH DESCRIPTOR:1111
A WINS
TIME ELAPSED       0.16

PATH NO.          2   PATH DESCRIPTOR:11101
B WINS
TIME ELAPSED       0.04

PATH NO.          3   PATH DESCRIPTOR:111CC
TIE
TIME ELAPSED       0.02

PATH NO.          4   PATH DESCRIPTOR:11C1
A WINS
TIME ELAPSED       0.02
```

Listing 3. Computer Output of the Driver System for the Program of Listings 1 and 2.

and so on for the remaining 96 paths.

6.0 Results

6.1 Introduction

In this section we shall attempt to discuss the usefulness of Type 1.5 drivers for forced testing, briefly comparing their use with other widely used testing strategies, and describing the efficiency and deficiencies of the present implementation.

6.2 Automatic Analysis and Forced-Execution

As shown in Section 4.3, even for a simple program like that of Figure 4.3, path identification and enumeration is not a trivial problem; it is time consuming and error prone. In such cases, the algorithm described in Section 5.4 has great advantages, because it not only identifies all paths and properly labels them, but its implementation as a path-analyzer program constructs all binary path descriptors. Hence, it is not necessary to spend time in designing data sets to exercise all paths (which is often unsuccessful). The driver system takes the subject program as the input; the analyzer creates the descriptor for each run or path; the translator modifies the code; and the modified program runs through all its paths as many times as there are paths.

6.3 Comparison between Manual and Automatic Testing

Consider again the program of Listing 1 and Figure 4.3. We will establish a timetable to compare results and time spent by testing with either input data or with the driver system. We begin by saying that it took 30 minutes to design a successfully compiled program. We then test the program manually and automatically(*).

Manual Testing With Data	Automatic Testing With The Driver
• Design of a data set to exercise some paths: 10 minutes	• Path analyzer, 1.13 min, 450K core
• Running of the program through these paths: 0.01 minutes	• Translator: 0.27 minutes
	• Translated subject: 0.04 minutes all paths
• Results: program is OK	• Results: one bug was discovered

The bug appeared in the form of two contradictory outputs; an erroneous ELSE clause was subsequently fixed in the subject program.

(*) These operations were carried out on the IBM 360/65 system at the Polytechnic Institute of New York, Brooklyn.

It appears that no particular penalty in running time has to be paid to let a program run through all its paths. In fact, a program with repetitive DO-loops may very well run faster through the driver than with some manual testing, because extra saving in running time is achieved by not letting loops run more than twice. This saving may become consistent for programs with many nested loops.

Since it is not necessary to design any testing data, the programmer saves time and effort for manual debugging. There is, moreover, no guarantee that testing with data achieves the result, as this case shows: the erroneous path had never been reached using the test data.

However, a strategy which is currently being researched would allow a user to supply testing data to exercise paths of interest, as well as inhibitors to suppress force-traversal of unwanted paths. The driver would automatically keep track of naturally-executed paths and resume control to force-traverse all remaining paths upon exhaustion of the testing data set.

6.4 Efficiency of the Driver

The path analyzer program is a literal implementation of the recursive algorithm described in Section 5.4.1. As such, it is slow and inefficient, and requires a large amount of computer memory. This is due to the fact that each time a recursive procedure, which is still open, calls itself either directly or indirectly, the PL/1 system reallocates new space for variables in main storage. The scanned subject program is stored as a string and in PL/1, unfortunately, a string of maximum length is allocated even if the string is declared with a varying length attribute. As a result, the path analyzer wastes a large number of bytes. This wasted memory is recuperated only when the last recursive call terminates, that is, at the point when the recursion stack begins to pop.

The problem has, in fact, been avoided by implementing the algorithm for path analysis in LISP, a language ideally suited for recursive function and string (i.e., list) manipulations. No prohibitive memory is needed, since the calling-by-value and the pointer mechanism uses space only when needed. However, the LISP interpreter is not widely available and is, in general, very slow compared to compilers for other high level languages.

One solution to the efficiency of the analyzer we are considering would be to modify the recursive algorithm in PL/1. Instead of storing in the stack the character string resulting from an intermediate computation, we would operate always on one copy of the program string and store the present scanning location, i.e., a one-word pointer. Hence, we would achieve a saving of 32K-bytes versus one word, which is close to an order of magnitude of 4. A drastic saving in memory space would be realized, and simultaneously some techniques to speed up the algorithm could be added.

The translator program is essentially built around string manipulating built-in procedures and is therefore fast. No doubt, however, it could be further optimized.

The translated subject program will run through all its paths. In the previous section we pointed out that this may be achieved at the price of some penalty in running time. It is worth mentioning, however, that even if this were so, the advantage provided by an automatic mechanism which is guaranteed to exercise all paths in a subject program outweighs some possible disadvantages of extra running time.

6.5 Restrictions on the Subject Program and Limitations of the Driver System.

Our programs implement a driver system which forces execution of a subject program written in PL/1 through all its possible paths. Almost all PL/1 features can be used in the design of the subject program, with only a few restrictions. Some of these restrictions are caused by our computer installation and some others have been introduced for ease of design of the system. Some are tolerable or even welcome since they encourage good programming style, but some are undesirable and will hopefully be removed in the near future.

We begin by mentioning that, as shown in Section 3.3, the driver will force execution of some paths that could never be reached by natural execution, hence creating error messages for non-existent errors. This is a flaw of the Type 1 testing model and could conceivably only be solved by a hierarchically higher model. It can be demonstrated rigorously that the determination by static analysis of unreachable paths is an impossible problem. In order to see this with an example, assume that within certain lines of code an algorithm is called for the computation of one of the values used in the next decider. Before execution it is unknown whether the algorithm will terminate or not (halting problem). This example shows that it is, in theory, impossible to exercise a path, even at level 1. Our driver discontinues a path if it takes longer than a certain time, thus setting a time bound to avoid the problem altogether. Possibly, only those impossible paths depending on assignments of known values to the decider variables can be identified and possibly only a subset of them could be found by a static analysis similar to the one of Figure 3.1. In the meantime, we must regard this problem as not a highly important one; the user can in fact quickly identify these paths from the driver's output.

We now describe a few implementation restrictions. Our version of PL/1 does not allow strings (hence, subject programs) with more than 32767 bytes, i.e., about 400 cards. Without modifying our local definition of PL/1, this could be solved by a more complex scanning algorithm operating upon arrays of strings, memory size permitting.

Ease in program design of the analyzer requires the exclusion, in the subject program, of any variable or identifier named IF, THEN, ELSE, DO, END, BEGIN, TO, BY and WHILE. Similarly, multiple clause DO-groups (where clauses are separated by commas), and multiple closure with a labeled END (as permitted by PL/1) have to be avoided. However, use of these PL/1 features causes confusion, and often programmers are independently encouraged not to use them.

Moreover, although it is a minor drawback that the driver cannot handle GO TO's (they can always be avoided in structured programs, and almost always with profit), it is more serious that it cannot cope, at this stage of development, with branching to subroutines. Consistent with the theory behind Type 1 testing, we plan to remove this restriction by allowing subroutines which the driver would enter and leave without any force-traversal mechanism. A subroutine would have to be tested individually by the driver to check its paths. It is unclear how a static path analyzer should cope with recursive procedures, because the path structure of a recursive program is unknown before execution time. In fact, no graph or flowchart representation exists for a recursive algorithm. Let us say that any recursive function can always, though often with a lot of effort, be made non-recursive. Besides, recursive programs are rarely found in the real world of programming.

It is relevant to note that any structured program can be written with the constructs allowed by our driver. Therefore, except for its inability to handle procedures (to be removed soon), the driver system is able enough to accept any well written PL/1 program to be tested.

7. Summary and Conclusions

We conclude by describing a practical application for a Type 1.5 testing model.

Our driver system could advantageously be integrated into an operating system. A program would initially be compiled to catch syntax errors. Upon successful compilation, it would be submitted to the driver system for forced execution. Hence, another set of errors, appearing at execution time, could be eliminated prior to the definitive testing with real data, or a strategy intermingling natural and forced execution could be implemented.

We would like to recall once more that our effort, although directed mainly toward PL/1 programs, can be extended to other programming languages. We hope, therefore, that our driver techniques represent a step toward automatic debugging.

As a final conclusion, we note that:

1. Although the area of testing is a difficult one, this report has developed several quantitative models and approaches to aid research progress in the field.

2. A quantitative way of describing and categorizing different types of tests has been developed which may aid discussion and characterization of tests.

3. A system of algorithms to perform automatic Type 1.5 testing has been implemented and described in detail. It was shown that such a driver is feasible and advantageous. An explicit computation of the number of paths in a flowgraph was carried out analytically. An example has been

submitted to the driver system and run through the driver. The testing model has been defined, researched and fully implemented. Although the model has already proved itself a useful tool, it is hoped that it will clarify and further stimulate research in this area.

References

(1) Denis L. Baggi and Martin L. Shooman, "An Automatic Driver for Pseudo-Exhaustive Software Testing", Proceedings of the CompCon, Feb. 28-March 3, 1978, San Francisco, IEEE No. 78CH1328-4C, p. 278.

(2) Denis L. Baggi and Martin L. Shooman, "Software Test Models and the Implementation of Associated Test Drivers", Summary of Technical Progress, Software Modeling Studies, SRS 114, POLY EE 78-052, Jan. 1, 1978-Dec 31, 1978.

(3) John R. Brown, "Testing Strategies for Software Reliability Assessment", Advanced Defense Systems, TRW Systems R2/2062, Redondo Beach, Ca.

(4) Robert S. Boyer, Bernard Elspas, Karl N. Levitt, "Select - a Formal System for Testing and Debugging Programs by Symbolic Execution", Proceedings of the International Conference on Reliable Software, Apr. 21-23, 1975, Los Angeles, IEEE

(5) Lori A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, Sept. 1976.

(6) James C. King, "Symbolic Execution and Program Testing", Proceedings of the IEEE Computer Software and Applications Conference, Nov. 8-11, 1977, Chicago.

(7) John A. Darringer & James C. King, "Applications of Symbolic Execution to Program Testing", Res. Rep. RC6965(#29464), 12/1/1977, IBM T.J. Watson Res. Center, Yorktown Heights, N.Y. See also IEEE Computer, April 1978.

(8) K.W. Krause, R.W. Smith, M.A. Goodwin, "Optimal Testing Through Automated Network Analysis", Record 1973 IEEE Symposium on Computer Software Reliability, April 30-May 2, 1973, New York, PIB.

(9) Myron Lipow, "Application of Algebraic Methods to Computer Program Analysis", TRW Systems Group, 1973.

(10) E.F. Miller, "Automated Generation of Testcase Datasets", Proceedings of the IEEE Computer Software and Applications Conference, Chicago, No. 8-11, 1977

(11) M.R. Paige and E.E. Balkovich, "On Testing Programs", IEEE Symposiums on Computer Software Reliability, Apr. 30-May 2, 1973, New York.

(12) David J. Panzl, "Automatic Software Test Drivers", IEEE Computer, April 1978.

(13) C.V. Ramamoorthy and S.F. Ho, "Testing Large Software with Automated Software Evaluation Systems", Proceedings of the IEEE International Conference on Reliable Software, Apr. 21-23, Los Angeles.

(14) Martin L. Shooman, "Probabilistic Models for Software Reliability Predictions", in Computer System Performance, Freiberger Ed., 1971, Academic Press, New York.

(15) Martin L. Shooman, "Meaning of Exhaustive Software Testing", Report PINY EE/IP 74-006/EER/106, Jan. 1974, Polytechnic Institute of New York.

(16) Automated Unit Test (AUT) Program Description Operation Manual. IBM Installed User Program Number 5796-PEC, August 1975.

(17) User Information for the Interactive Automated Test Data Generator (ATDG) System, JSC-10832 Rev 1, NASA, Houston, Texas.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.